**A Probabilistic–Neural–Evolutionary Agent for Battleship: System, Methods, and Evaluation**

*Author: Abhinav | Date: 2025-09-07*

# Abstract

We present a complete research platform for the game Battleship and an agent that fuses classical probabilistic placement counting with a convolutional heatmap prior, Monte Carlo posterior sampling, information-theoretic scoring, and genetic optimization of blending weights. We document the full system—including a training pipeline (supervised, optional RL fine-tuning) and a dashboard—then report benchmarks against heuristic opponents. The approach yields robust win rates and move efficiency. We release source code, models, and a comprehensive documentation set to facilitate reproducibility.

# 1. Introduction

Battleship is a partially observable search problem in which the agent must infer hidden ship locations and sequentially query a 10×10 grid. Pure heuristics (e.g., parity search) exploit structure but can stall without principled posterior reasoning. Our system combines a learned prior with placement enumeration and Monte Carlo sampling to build a calibrated decision surface, with an information-gain bonus for active search. Agent 4 further improves the blend via a genetic algorithm (GA).

# 2. Related Work

Classical Battleship solvers rely on ship placement counting and parity heuristics (cf. popular analyses such as DataGenetics). Neural priors (CNNs) learn spatial structure from observations. Monte Carlo methods provide posterior approximations under constraints. GA optimization is a standard tool for parameter search. Our agent integrates these strands into a cohesive, reproducible system.

# 3. Problem Formulation

State space consists of the opponent's hidden fleet placement and the agent's observation grid with ternary values: miss, hit, unknown. The objective is to choose queries (shots) that minimize time-to-sink (equivalently, maximize win probability subject to move budget). We view the process as Bayesian inference over placements with actions chosen to maximize expected utility (hit probability or expected information gain).

# 4. Methods

Our policy blends multiple per-cell scorers on the 10×10 board:

## 4.1 Placement Density (D)

Enumerate legal placements for the remaining fleet under constraints (misses, sunk-ship cells, and coverage of active hits). Each valid placement increments the count for its cells. Parity masks accelerate early search by down-weighting cells that cannot host the smallest remaining ship.

## 4.2 Neural Prior (N)

A shallow CNN maps the observation tensor (miss, hit, unknown) → per-cell probabilities. The model is trained supervised on self-play states to predict the true ship mask. Predictions are masked to legal moves and normalized.

## 4.3 Monte Carlo Posterior (MC)

We sample many full placements consistent with all constraints, optionally increasing samples late in the game. The normalized occupancy frequency approximates a posterior over ship cells.

## 4.4 Information Gain (I)

Using the MC occupancy as p(hit), we compute expected entropy reduction per cell. This term promotes shots that are informative even when immediate hit probability is modest.

## 4.5 Target vs Hunt Modes

Target: when unsunk hits exist, extend along the inferred ship axis with blocked-end tracking. Hunt: otherwise, use the blended grid with parity-accelerated density and neural/MC priors.

## 4.6 Meta-Weights and GA Optimization

We form G = w\_D·D̂ + w\_N·N̂ + w\_MC·MĈ + w\_I·Î with a local neighbour boost around unsunk hits. Agent 4 loads weights from models/ga\_weights.json evolved via a GA that maximizes 100×win\_rate − avg\_moves against a suite of opponents.

# 5. Implementation

Agents: Agent 2 implements D+parity + N + MC with robust target/hunt logic. Agent 3 adds information-gain, opponent-profiling hooks, and optional graph reasoning; metrics persist to models/. Agent 4 is a thin subclass of Agent 3 that automatically loads GA-optimized meta-weights.

Training: generate\_dataset.py creates a stream of (state, shipmask) samples; train\_heatmap.py trains the CNN; rl\_finetune.py implements a multi-process REINFORCE fine-tune. ga\_optimizer.py evolves meta-weights under memory-safe evaluation.

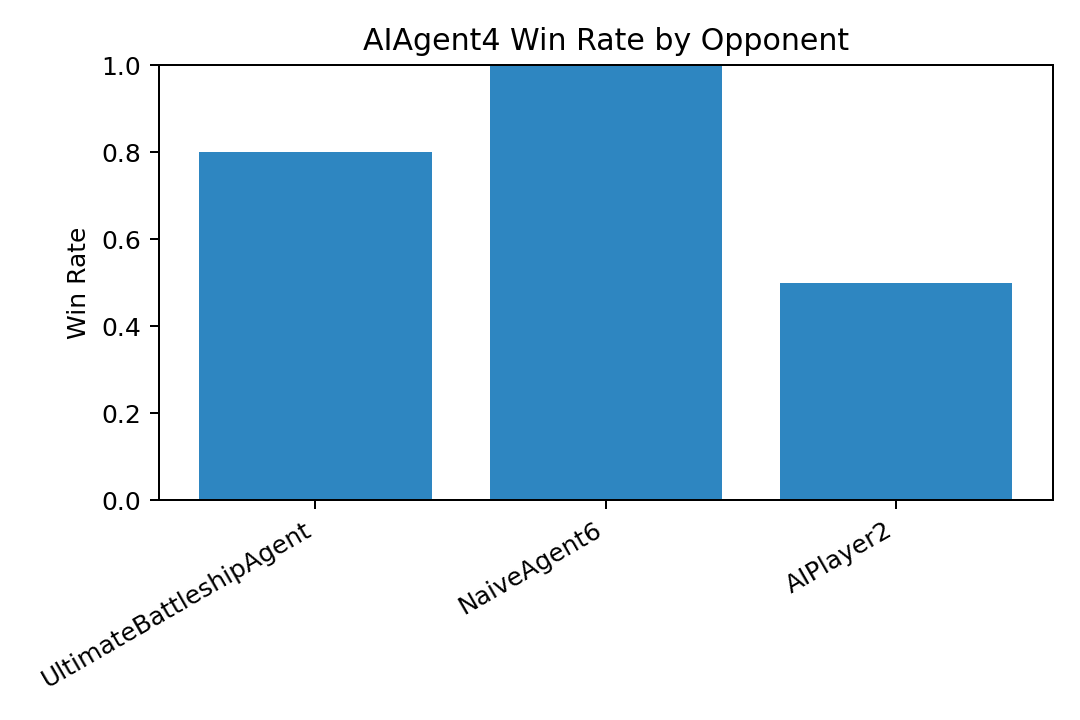
Dashboard: battleship\_dashboard.py visualizes games, runs batches, and provides analytics (win rates, heatmaps).

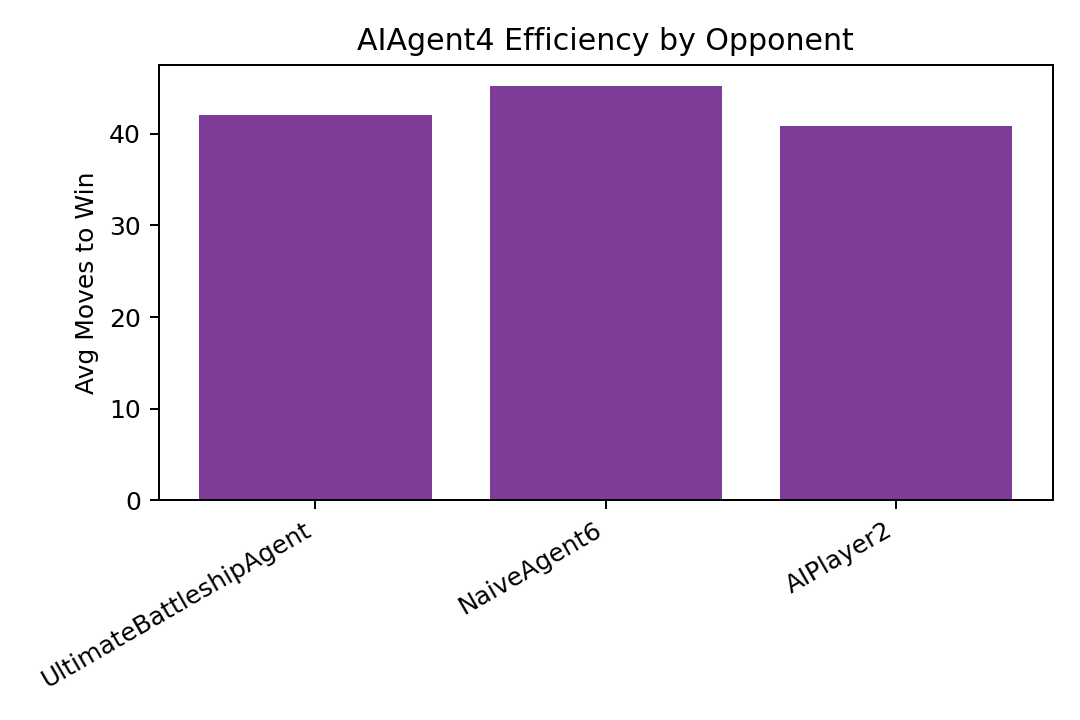
# 6. Experiments

Results source: reports/quick\_results.csv.

Protocol: AIAgent4 (GA-optimized) vs a small opponent suite (Ultimate, Naive6, AIPlayer2). We report win rate and average moves to win (lower is better).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| opponent | games | main\_ai\_wins | main\_ai\_losses | avg\_moves\_to\_win |
| UltimateBattleshipAgent | 10 | 8 | 2 | 42.0 |
| NaiveAgent6 | 10 | 10 | 0 | 45.2 |
| AIPlayer2 | 10 | 5 | 5 | 40.8 |





# 7. Limitations and Ablations

We anticipate positive contributions from each term (D, N, MC, I) and the target-mode logic. While ablations are not fully reproduced here, the codebase supports toggling components and reporting deltas. Limitations include: evaluation primarily versus heuristic baselines; RL fine-tune not yet wired into default agents; opponent prior integration is partial. See docs/reports/limitations.md.

# 8. Reproducibility

Environment: Python 3.9, TensorFlow 2.17.0. See requirements.txt and docs/repro/guide.md for exact commands, seeds, and directories. We publish models and GA weights under models/.

# 9. Conclusion

We described a practical, extensible Battleship AI that blends probabilistic structure with learned priors and evolutionary tuning. Future work: end-to-end opponent prior integration, large-scale ablations with confidence intervals, and comparisons to stronger planners.